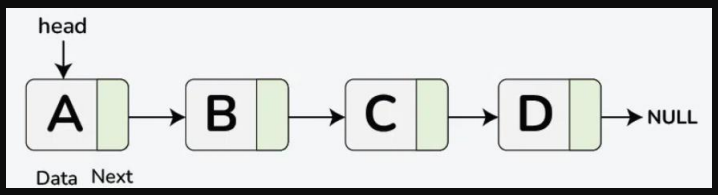
Feature	Linked List	Array
Data Structure	Non-contiguous	Contiguous
Memory Allocation	Typically allocated one by one to individual elements	Typically allocated to the whole array
Insertion/Deletion	Efficient	Inefficient
Access	Sequential	Random

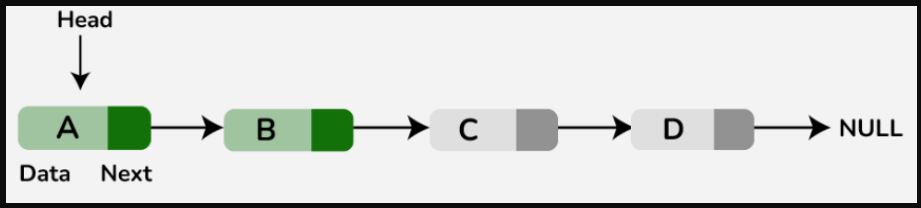
Singly Linked List




```
// Definition of a Node in a singly linked list
struct Node {
    int data;
    Node* next;
    Node(int data)
    {
        this->data = data;
        this->next = nullptr;
    }
};
```

operation on singly Linked list are-

1. **Traversal -------**



```cpp
// (Iterative Approach)
Tabnine | Edit | Test | Explain | Document
void traverseList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}
//  (Recursive Approach)
Tabnine | Edit | Test | Explain | Document
void traverseList(Node* head) {
    if (head == nullptr) {
        cout << endl;
        return;
    }
    cout << head->data << " ";
    traverseList(head->next);
}
```

2. Searching----------

1-

```cpp
// (Iterative Approach) - O(N) Time and O(1) Space
Tabnine | Edit | Test | Explain | Document
bool searchKey(Node* head, int key) {
    Node* curr = head;
    while (curr != NULL) {
        if (curr->data == key)
            return true;
        curr = curr->next;
    }
    return false;
}
```

2-

```cpp
// Search an element in a Linked List (Recursive
Tabnine | Edit | Test | Explain | Document
bool searchKey(struct Node* head, int key) {
    if (head == NULL)
        return false;
    if (head->data == key)
        return true;
    return searchKey(head->next, key);
}
```

3. Length----------

```
// Iterative Approach
Tabnine | Edit | Test | Explain | Document
int countNodes(Node* head) {
    int count = 0;
    Node* curr = head;
    while (curr != nullptr) {
        count++;
        curr = curr->next;
    }
    return count;
}
// Recursive Approach
Tabnine | Edit | Test | Explain | Document
int countNodes(Node* head) {
    if (head == NULL) {
        return 0;
    }
    return 1 + countNodes(head->next);
}
```

4. Insertion

1-

```
Node* insertAtFront(Node* head, int new_data)
{
    Node* new_node = new Node(new_data);
    new_node->next = head;
    return new_node;
}
```

2-

```
Node* insertAtEnd(Node* head, int new_data) {
    Node* new_node = new Node(new_data);
    if (head == nullptr) {
        return new_node;
    }
    Node* last = head;
    while (last->next != nullptr) {
        last = last->next;
    }
    last->next = new_node;
    return head;
}
```

3-

```
Node *insertPos(Node *head, int pos, int data) {
    if (pos < 1)
        return head;
    if (pos == 1) {
        Node *newNode = new Node(data);
        newNode->next = head;
        return newNode;
    }
    Node *curr = head;
    for (int i = 1; i < pos - 1 && curr != nullptr; i++) {
        curr = curr->next;
    }
    if (curr == nullptr)
        return head;
    Node *newNode = new Node(data);
    newNode->next = curr->next;
    curr->next = newNode;
    return head;
}
```

5. Deletion

```
// 1.Deletion at beginning
Tabnine | Edit | Test | Explain | Document
Node* deleteHead(Node* head) {
    if (head == nullptr)
        return nullptr;
    Node* temp = head;
    head = head->next;
    delete temp;
    return head;
}
```

```
// 2.Deletion at end
Tabnine | Edit | Test | Explain | Document
Node* removeLastNode(struct Node* head)
{
    if (head == nullptr) {
        return nullptr;
    }
    if (head->next == nullptr) {
        delete head;
        return nullptr;
    }
    Node* second_last = head;
    while (second_last->next->next != nullptr) {
        second_last = second_last->next;
    }
    delete (second_last->next);
    second_last->next = nullptr;
    return head;
}
```

```
// 3.Delete a Linked List node at a given position
Tabnine | Edit | Test | Explain | Document
Node* deleteNode(Node* head, int position)
{
    Node* prev;
    Node* temp = head;
    if (temp == NULL)
        return head;
    if (position == 1) {
        head = temp->next;
        free(temp);
        return head;
    }
    for (int i = 1; i != position; i++) {
        prev = temp;
        temp = temp->next;
    }
    if (temp != NULL) {
        prev->next = temp->next;
        free(temp);
    }
    else {
        cout << "Data not present\n";
    }
    return head;
}
```

6. Modify

```
Node* reverse(Node* head) {
    Node* prev = nullptr;
    Node* curr = head;
    Node* next = nullptr;

    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    return prev;
}
```

```cpp
Node* modifyTheList(Node* head) {
    if (!head->next) {
        return head;
    }
    Node* slow = head;
    Node* fast = head;
    Node* mid;
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    mid = slow;
    Node* reversedList = mid->next;
    mid->next = nullptr;
    reversedList = reverse(reversedList);
    Node* curr1 = head;
    Node* curr2 = reversedList;
    vector<int> firstHalf, secondHalf;
    while (curr1 != nullptr) {
        firstHalf.push_back(curr1->data);
        curr1 = curr1->next;
    }
    while (curr2 != nullptr) {
        secondHalf.push_back(curr2->data);
        curr2 = curr2->next;
    }
    for (int i = 0; i < secondHalf.size(); i++) {
        int x = firstHalf[i];
        firstHalf[i] = secondHalf[i] - x;
        secondHalf[i] = x;
    }
    curr1 = head;
    for (int val : firstHalf) {
        curr1->data = val;
        curr1 = curr1->next;
    }
    curr2 = reversedList;
    for (int val : secondHalf) {
        curr2->data = val;
        curr2 = curr2->next;
    }
    mid->next = reverse(reversedList);
    return head;
}
```

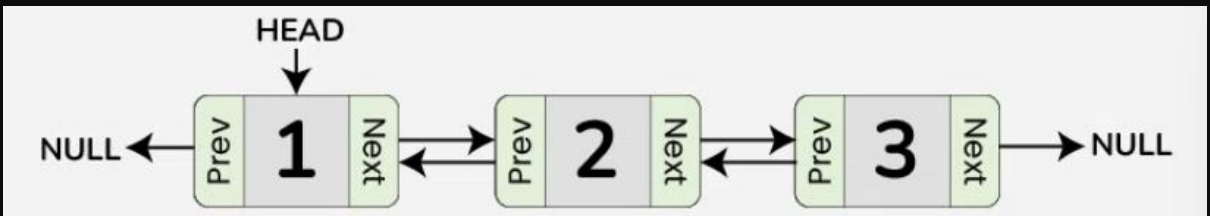
7. Reversing

```cpp
// Using Iterative Method - O(n) Time and O(1) Space
Tabnine | Edit | Test | Explain | Document
Node *reverseList(Node *head) {
    Node *curr = head, *prev = nullptr, *next;
    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

```cpp
// Using Recursion - O(n) Time and O(n) Space
Tabnine | Edit | Test | Explain | Document
Node *reverseList(Node *head) {
    if (head == NULL || head->next == NULL)
        return head;
    Node *rest = reverseList(head->next);
    head->next->next = head;
    head->next = NULL;
    return rest;
}
```

```cpp
// Using Stack - O(n) Time and O(n) Space
Tabnine | Edit | Test | Explain | Document
Node* reverseList(Node* head) {
    stack<Node*> s;
    Node* temp = head;
    while (temp->next != NULL) {
        s.push(temp);
        temp = temp->next;
    }
    head = temp;
    while (!s.empty()) {
        temp->next = s.top();
        s.pop();
        temp = temp->next;
    }
    temp->next = NULL;
    return head;
}
```

Doubly Linked List



Node creation---

```
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int d) {
        data = d;
        prev = next = nullptr;
    }
};
```

Operations---

1. Traversal----Types of Traversal in Doubly Linked List--Forward Traversal, Backward Traversal

Forward traversal-

```
// 1. Iterative Approach for Forward Traversal
Tabnine | Edit | Test | Explain | Document
void forwardTraversal(Node *head) {
    Node *curr = head;
    while (curr != nullptr) {
        cout << curr->data << " ";
        curr = curr->next;
    }
    cout << endl;
}
// 2. Recursive Approach for Forward Traversal
Tabnine | Edit | Test | Explain | Document
void forwardTraversal(Node *head) {
    if (head == nullptr)
        return;
    cout << head->data << " ";
    forwardTraversal(head->next);
}
```

Backward Traversal

```
// 3. Iterative Approach for Backward Traversal
Tabnine | Edit | Test | Explain | Document
void backwardTraversal(Node* tail) {
    Node* curr = tail;
    while (curr != nullptr) {
        cout << curr->data << " ";
        curr = curr->prev;
    }
}
// 4. Recursive Approach for Backward Traversal
Tabnine | Edit | Test | Explain | Document
void backwardTraversal(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    backwardTraversal(node->prev);
}
```

2. Insertion –

```
// 1.Insertion at Beginning
Tabnine | Edit | Test | Explain | Document
Node *insertAtFront(Node *head, int new_data) {
    Node *new_node = new Node(new_data);
    new_node->next = head;
    if (head != NULL)
        head->prev = new_node;
    return new_node;
}
```

```
// 2.Insertion at End
Tabnine | Edit | Test | Explain | Document
Node *insertEnd(Node *head, int new_data) {
    Node *new_node = new Node(new_data);
    if (head == NULL) {
        head = new_node;
    }
    else {
        Node *curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }
        curr->next = new_node;
        new_node->prev = curr;
    }
    return head;
}
```

```
// 3.Insertion at Specific Position
Tabnine | Edit | Test | Explain | Document
Node *insertAtPosition(Node *head, int pos, int new_data) {
    Node *new_node = new Node(new_data);
    if (pos == 1) {
        new_node->next = head;
        if (head != NULL)
            head->prev = new_node;
        head = new_node;
        return head;
    }
    Node *curr = head;
    for (int i = 1; i < pos - 1 && curr != NULL; ++i) {
        curr = curr->next;
    }
    if (curr == NULL) {
        cout << "Position is out of bounds." << endl;
        delete new_node;
        return head;
    }
    new_node->prev = curr;
    new_node->next = curr->next;
    curr->next = new_node;
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
    return head;
}
```

3. length----

```
// Approach - Using While Loop – O(n) Time and O(1) Space
Tabnine | Edit | Test | Explain | Document
int findSize(Node *curr) {
    int size = 0;
    while (curr != NULL) {
        size++;
        curr = curr->next;
    }
    return size;
}
// Approach - Using Recursion - O(n) Time and O(n) Space
Tabnine | Edit | Test | Explain | Document
int findSize(Node* head) {
    if (head == NULL)
        return 0;
    return 1 + findSize(head->next);
}
```

4. deletion----

```
// 1.Deletion at beginning
Tabnine | Edit | Test | Explain | Document
Node* deleteHead(Node* head) {
    if (head == nullptr)
        return nullptr;
    Node* temp = head;
    head = head->next;
    delete temp;
    return head;
}
```

```
// 2.Deletion at End--Time Complexity: O(N),Auxiliary Space: O(1)
Tabnine | Edit | Test | Explain | Document
Node *delLast(Node *head) {
    if (head == NULL)
        return NULL;
    if (head->next == NULL) {
        delete head;
        return NULL;
    }
    Node *curr = head;
    while (curr->next != NULL)
        curr = curr->next;
    curr->prev->next = NULL;
    delete curr;
    return head;
}
```

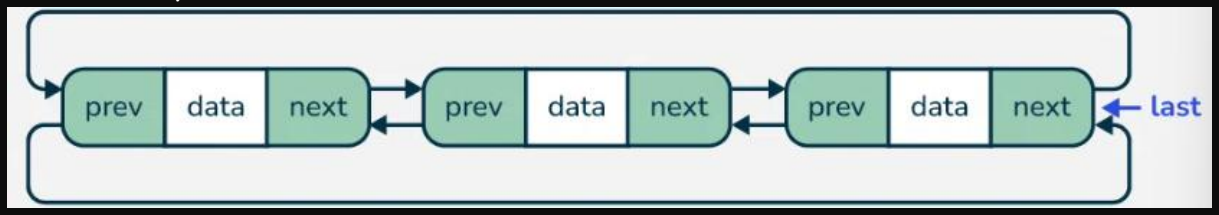
```
// 3.Deletion at at a given position--Time Complexity: O(N),Auxiliary Space: O(1)
Tabnine | Edit | Test | Explain | Document
Node * delPos(Node* head, int pos) {
    if (head == NULL)
        return head;

    Node * curr = head;
    for (int i = 1; curr != NULL && i < pos; ++i) {
        curr = curr -> next;
    }
    if (curr == NULL)
        return head;
    if (curr -> prev != NULL)
        curr -> prev -> next = curr -> next;
    if (curr -> next != NULL)
        curr -> next -> prev = curr -> prev;
    if (head == curr)
        head = curr -> next;
    delete curr;
    return head;
}
```

Circular Linked List

Types of Circular Linked Lists

1. **Circular Singly Linked List**

2. **Circular Doubly Linked List**



Operations on the Circular Linked list

Node creation-

```cpp
#include <iostream>
using namespace std;
struct Node{
    int data;
    Node *next;
    Node(int value){
        data = value;
        next = nullptr;
    }
};
```

1. Insertion-

```cpp
// Insertion in an empty List in the circular linked list
Tabnine | Edit | Test | Explain | Document
Node *insertInEmptyList(Node *last, int data){
    if (last != nullptr) return last;
    Node *newNode = new Node(data);
    newNode->next = newNode;
    last = newNode;
    return last;
}
```

```cpp
// Insertion at the beginning in circular linked list
Tabnine | Edit | Test | Explain | Document
Node* insertAtBeginning(Node* last, int value){
    Node* newNode = new Node(value);
    if (last == nullptr) {
        newNode->next = newNode;
        return newNode;
    }
    newNode->next = last->next;
    last->next = newNode;
    return last;
}
// Insertion at the end in circular linked list
Tabnine | Edit | Test | Explain | Document
Node *insertEnd(Node *tail, int value)
{
    Node *newNode = new Node(value);
    if (tail == nullptr){
        tail = newNode;
        newNode->next = newNode;
    }
    else{
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }
    return tail;
}
```

```cpp
// Insertion at specific position in circular linked list
Tabnine | Edit | Test | Explain | Document
Node *insertAtPosition(Node *last, int data, int pos){
    if (last == nullptr){
        if (pos != 1){
            cout << "Invalid position!" << endl;
            return last;
        }
        Node *newNode = new Node(data);
        last = newNode;
        last->next = last;
        return last;
    }
    Node *newNode = new Node(data);
    Node *curr = last->next;
    if (pos == 1){
        newNode->next = curr;
        last->next = newNode;
        return last;
    }
    for (int i = 1; i < pos - 1; ++i) {
        curr = curr->next;
        if (curr == last->next){
            cout << "Invalid position!" << endl;
            return last;
        }
    }
    newNode->next = curr->next;
    curr->next = newNode;
    if (curr == last) last = newNode;
    return last;
}
```

2. Deletion

```cpp
// 1. Delete the first node in circular linked list
Tabnine | Edit | Test | Explain | Document
Node* deleteFirstNode(Node* last) {
    if (last == nullptr) {
        cout << "List is empty" << endl;
        return nullptr;
    }
    Node* head = last->next;
    if (head == last) {
        delete head;
        last = nullptr;
    } else {
        last->next = head->next;
        delete head;
    }
    return last;
}
```

```cpp
// 2. Delete a specific node in circular linked list
Tabnine | Edit | Test | Explain | Document
Node* deleteSpecificNode(Node* last, int key) {
    if (last == nullptr) {
        cout << "List is empty, nothing to delete." << endl;
        return nullptr;
    }
    Node* curr = last->next;
    Node* prev = last;
    if (curr == last && curr->data == key) {
        delete curr;
        last = nullptr;
        return last;
    }
    if (curr->data == key) {
        last->next = curr->next;
        delete curr;
        return last;
    }
    while (curr != last && curr->data != key) {
        prev = curr;
        curr = curr->next;
    }
    if (curr->data == key) {
        prev->next = curr->next;
        if (curr == last) {
            last = prev;
        }
        delete curr;
    } else {
        cout << "Node with data " << key
          << " not found." << endl;
    }
    return last;
}
```

```
// 3. Deletion at the end of Circular linked list
Tabnine | Edit | Test | Explain | Document
Node* deleteLastNode(Node* last) {
    if (last == nullptr) {
        cout << "List is empty, nothing to delete." << endl;
        return nullptr;
    }
    Node* head = last->next;
    if (head == last) {
        delete last;
        last = nullptr;
        return last;
    }
    Node* curr = head;
    while (curr->next != last) {
        curr = curr->next;
    }
    curr->next = head;
    delete last;
    last = curr;
    return last;
}
```

3. Searching

```
void find(int key)
{
    Node* temp = head;
    int f = 0;
    if (head == nullptr) {
        cout << "List is empty" << endl;
    }
    else {
        do {
            if (temp->data == key) {
                cout << key << " Found" << endl;
                f = 1;
                break;
            }
            temp = temp->next;
        } while (temp != head);

        if (f == 0) {
            cout << key << " Not Found" << endl;
        }
    }
}
```